A linear concurrent constraint approach for the automatic verification of access permissions

Carlos Olarte, Camilo Rueda, Elaine Pimentel, Nestor Cataño

A Linear Concurrent Constraint approach for the automatic verification of access permissions

Carlos Olarte, Camilo Rueda
Dept. de Electrónica y Ciencias de la Computación.
Pontificia Universidad Javeriana-Cali, Colombia
{caolarte, crueda}@cic.puj.edu.co

Elaine Pimentel
Universidad del Valle, Colombia.
Universidade Federal de Minas Gerais, Brasil
elaine @mat.ufmg.br

Néstor Cataño
The University of Madeira, Portugal
ncatano@uma.pt

Abstract

A recent trend in object oriented programming languages is the use Access Permissions (AP) as abstraction to control concurrent executions. AP define a protocol specifying how different references can access the mutable state of objects. Although AP simplify the task of writing concurrent code, an unsystematic use of permissions in the program can lead to subtle problems. This paper presents a Linear Concurrent Constraint (`lcc`) approach to verify AP annotated programs. We model AP as constraints (i.e., formulas in logic) in an underlying constraint system, and we use entailment of constraints to faithfully model the flow of AP in the program. We verify relevant properties about programs by taking advantage of the declarative interpretation of `lcc` agents as formulas in linear logic. Properties include deadlock detection, program correctness (whether programs adhere to their AP specifications or not), and the ability of methods to run concurrently. We show that those properties are decidable and we present a complexity analysis of finding such proofs. We implemented our verification and analysis approach as the Alcove tool, which is available on-line.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs; D.3.2 [*Language Classifications*]: Constraint and logic languages. Concurrent, distributed, and parallel languages.

General Terms Theory, Verification, Concurrency

Keywords Concurrent Constraint Programming, Access Permissions, Linear Logic, Verification

1. Introduction

Reasoning about concurrent programs is much harder than reasoning about sequential ones. Programmers often find themselves overwhelmed by the many subtle cases of thread interaction they must be aware of so as to decide whether a concurrent program is correct or not. Also, the need of finding the right level of thread atomicity, avoiding race conditions, coping with mutual exclusion requirements, guaranteeing deadlock freeness, make it very hard to design reliable concurrent software. Furthermore, the attempt to find errors through testing is doomed to failure because of the nondeterminism caused by thread scheduling.

This complexity of concurrent software is aggravated when software designers, wishing to take advantage of object oriented (OO) design strategies, use OO languages to write concurrent programs. The distribution of state in objects that might have multiple references (aliases), probing or modifying concurrently their *local* contents, contributes significantly to the complexity of sound concurrent program design. This potential data race situation occurs when a reading and a writing trace both access a shared memory location. This should be considered a program error since it gives rise to inconsistent executions paths. To cope with this problem, a simple strategy is to wrap each object access up in an atomic block. However, this negatively affects program performance. A better strategy would be for the programmer to lock just those objects that are actually shared among threads. It is very hard, however, to figure out which objects are to be shared and what locations are really protected by the locks simply by looking at the program text.

Languages like Æminium [18] and Plaid [19] propose a different strategy to concurrency based on *access permissions* (AP) [5] to objects. AP are descriptions about how various references to an object can coexist. They permit a direct control about the access to the mutable state of an object. Making the access to a shared mutable state explicit facilitates verification and it also permits parallelization of code. For instance, a *unique* AP, which describes the case when only one reference to a given object exists, enforces absence of interference and simplifies verification. On the other hand, a *shared* AP, which describes the case when an object may be accessed and modified by multiple references, allows for concurrent executions and makes verification trickier.

Although AP greatly help to devise static strategies for correct concurrent sharing of objects, the interactions resulting from dynamic bindings (e.g., aliasing of variables) might still lead to subtle difficulties. Indeed, it may happen that apparently correct permissions assignments in simple programs lead to deadlocks.

We propose a Linear Concurrent Constraint (`lcc`) [7] programming approach to the verification of AP annotated programs. In our approach, programs are interpreted as `lcc` agents that use constraints to keep information about AP, object references, object fields, and method calls. We use constraint entailment to verify compliance of methods and arguments to their AP based signatures. Furthermore, by exploiting the declarative view of `lcc` agents as logical formulas, we are able to analyze and verify programs. The proposed program verification includes (1) deadlock detection; (2) whether it is possible for methods to be executed concurrently

```
1  class stats {...}
2  class collection {
3    collection() none(this) ⇒ unq(this) {...}
4    sort() unq(this) ⇒ unq(this) {...}
5    print() imm(this) ⇒ imm(this) {...}
6    compStats(stats s) imm(this), unq(s) ⇒
                        imm(this), unq(s) {...}
7    removeDuplicates() unq(this) ⇒ unq(this){...}}
8  main() {
9    let collection c, stats s in
10     c := new collection()
11     s := new stats()
12     c.sort()
13     c.print()
14     c.compStats(s)
15     c.removeDuplicates()
16   end  }
```

Figure 1. Example of an Æminium program.

or not; and (3) whether annotations adhere to the intended semantics associated with AP or not.

The contributions of the this paper are four-fold (1) the definition of an elegant `lcc` semantics of AP for an object oriented concurrent programming language; (2) the definition of a decidable efficient verification procedure of non-recursive programs; (3) a complexity analysis of the effort required to verify a program; and (4) the implementation of the Alcove tool that automates our verification approach.

The rest of the paper is organized as follows. Section 2 presents the syntax of the AP based language used here and recalls `lcc`. Section 3 presents the model of AP as `lcc` agents. We also show how the proposed model is a runnable specification that allows users to observe the flow of program permissions. We implemented this models as the Alcove LCC Animator. Section 4 describes our approach to verify programs and its implementation as the Alcove LL prover. It also presents a complexity analysis about the proposed verification. Section 5 concludes the paper.

2. Preliminaries

2.1 Programs Syntax

Access permissions (AP) are abstractions describing how objects are accessed. Assume a variable x that points to the object o. The unique permission `unq` states that x is the sole reference to object o. The shared permission `shr` provides x with reading and modifying access to object o, which allows other references to o (called aliases) to exist and to read from it or to modify it. The immutable permission `imm` provides x with read-only access to o, and allows any other reference to object o to exist and to read from it. If x points to null, the permission `none` represents the fact that x is a null reference and it has no permission to access any object.

Figure 1 shows a program (taken and slightly modified from [18]) that operates over a collection of elements. Starting at line 8, the program creates an object of type $collection$ at Line 10 and an object of type $stats$ at line 11. The program sorts the collection c at line 12, and prints it at line 13. It computes some statistics at line 14, and removes duplicates from the collection at line 15. Lines 3-7 declare the signatures for the methods. The signature of class $collection$ constructor returns a unique reference to a new collection at line 3. Methods $sort$ and $removeDuplicates$ require a unique reference to the collection to exist and to return a unique permission to it. Method $compStats$ requires and returns an immutable (read-only) AP to the collection c and a unique AP to the parameter s.

Given these method signatures, the AP flow for the program is computed. Permissions can be produced and consumed. Hence, the unique permission returned by the constructor of class $collection$ is consumed by the call of $sort$. Once this method terminates, the unique permission is restored and split into two immutable permissions, and methods $print$ and $compStats$ can be executed concurrently. Once both methods have finished their execution, the immutable access permissions are joined back into a unique access permission, and the method $removeDuplicates$ can be executed.

The analyses presented in this paper considers a subset of Æminium [18], a concurrent-by-default object oriented programming language based on the above idea of AP (see Figure 2). Methods specify the required permissions for the caller ($p(\texttt{this})$) and for each argument ($\overline{p(y)}$) as well as the permissions restored to the environment when the method terminates ($p'(\texttt{this})$ and $\overline{p'(y)}$). Similarly for class constructors (CTR). We assume that in a call to a method (or constructor), the actual parameters are *references* (i.e., variables, object fields or `this`) and not arbitrary expressions. Since we have parameters by reference, we assume that the returned type is `void` and we omit it in the signature. For assignments we allow only statements of the form $r_l := r_r$, where the right and left hand side are references. Notice that we do not lose generality by imposing these syntactic restrictions since it is possible to unfold more general expressions by using local variables.

2.2 Linear `ccp`

Concurrent Constraint Programming (`ccp`) [17] is a model for concurrency that combines the traditional operational view of process calculi with a declarative view based on logic. This allows `ccp` to benefit from the large set of reasoning techniques of both process calculi and logic. Agents in `ccp` *interact* with each other by *telling* and *asking* information represented as *constraints* to a global store.

The basic constructs (processes) in `ccp` are: (1) the *tell* agent c, which adds the constraint c to the store, thus making it available to the other processes. Once a constraint is added, it cannot be removed from the store (i.e., the store grows monotonically). And (2), the *ask* process $c \rightarrow P$, which queries if c can be deduced from the information in the current store; if so, the agent behaves like P, otherwise, it remains blocked until more information is added to the store. In this way, ask processes define a simple and powerful synchronization mechanism based on entailment of constraints.

Linear Concurrent Constraint (`lcc`) [7] is a `ccp`-based calculus that considers constraint systems built from a fragment of Girard's intuitionistic linear logic (ILL) [8]. The move to a *linear discipline* permits ask agents to *consume* information (i.e., constraints) from the store.

Definition 1 (Linear Constraint Systems [7])**.** *A linear constraint system is a pair $(\mathcal{C}, \vdash)$ where $\mathcal{C}$ is a set of formulas (linear constraints) built from a signature Σ (a set of function and relation symbols), a denumerable set of variables $\mathcal{V}$ and the following ILL operators:* multiplicative *conjunction ($\otimes$) and its neutral element (1), the existential quantifier ($\exists$), the exponential bang (!) and the constant top ($\top$). Let Δ be a (possibly empty) subset of $\mathcal{C} \times \mathcal{C}$ defining the non-logical axioms of the constraint system (i.e, a theory). Then the entailment relation $\vdash$ is the least set containing Δ and closed by the rules of ILL (see Figure 3).*

We shall use $c, c', d, d' \ldots$ to denote elements in $\mathcal{C}$. We recall that $!\,c$ represents the arbitrary duplication of the resource c. The entailment $d \vdash c$ means that the information c can be deduced from the information represented by d.

$$
\begin{array}{lrcl}
\text{(programs)} & P & ::= & \langle \overline{CL}\ main \rangle \\
\text{(class decl.)} & CL & ::= & \texttt{class}\ cname\ \{\ \overline{F}\ \overline{M}\ \} \\
\text{(field decl.)} & F & ::= & cname\ fname \\
\text{(method decl.)} & M & ::= & meth(\overline{cname\ y})\ p(\texttt{this}), \overline{p(y)}\ \Rightarrow\ p'(\texttt{this}), \overline{p'(y)}\ \{s\} \\
 & CTR & ::= & cname(\overline{cname\ y})\ \texttt{none}(\texttt{this}), \overline{p(y)}\ \Rightarrow\ p'(\texttt{this}), \overline{p'(y)}\ \{s\} \\
\text{(main)} & main & ::= & \texttt{main}()\ \{s\} \\
\text{(references)} & r & ::= & x \mid x.fname \mid \texttt{this} \\
\text{(statements)} & s & ::= & \texttt{let}\ \overline{cname\ x}\ \texttt{in}\ s\ \texttt{end} \mid r_l := r_r \mid\ x.meth(\overline{r}) \mid x := \texttt{new}\ cname(\overline{r}) \mid s_1\ s_2\ ...\ s_n \\
\text{(permissions)} & p & ::= & \texttt{unq} \mid \texttt{shr} \mid \texttt{imm}
\end{array}
$$

Figure 2. Reduced Syntax of Æminium programs. $\overline{x}$ denotes a sequence of variables $x_1, ..., x_n$. This notation is similarly used for other syntactic categories.

$$
\begin{array}{ccc}
c \vdash c & \vdash 1 & \Gamma \vdash \top \\
\dfrac{\Gamma \vdash c}{\Gamma, 1 \vdash c} & \dfrac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} & \dfrac{\Gamma \vdash c_1 \quad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \\
\dfrac{\Gamma \vdash c[t/x]}{\Gamma \vdash \exists x.c} & \dfrac{\Gamma, c \vdash d \quad x \notin fv(\Gamma, d)}{\Gamma, \exists x.c \vdash d} & \dfrac{\Gamma, c \vdash d}{\Gamma, !\, c \vdash d} \\
\dfrac{\Gamma \vdash d}{\Gamma, !\, c \vdash d} & \dfrac{\Gamma, !\, c, !\, c \vdash d}{\Gamma, !\, c \vdash d} & \dfrac{!\,\Gamma \vdash d}{!\,\Gamma \vdash !\, d}
\end{array}
$$

Figure 3. Rules for the 1,⊗, ∃, ! fragment of the Intuitionistic Linear Logic (ILL). $fv(A)$ denotes the set of free variables of formula A. Γ, Δ denote set of formulas.

The Language of Processes. Similar to other ccp-based calculi, lcc, in addition to tell and ask agents, provides constructs for parallel composition, hidden of variables, non-deterministic choices and process definitions and calls.

Definition 2 (lcc agents [7])**.** *Agents in lcc are built from constraints in the underlying linear constraint system, following the syntax below.*

$$P, Q, ... ::= c \mid \forall \overline{x}(c \to P) \mid P \parallel Q \mid \exists x(P) \mid P + Q \mid p(\overline{x})$$

Tell Agent. Given a store d, the *tell* agent c adds c to d producing the new store $d \otimes c$.

Linear ask agent. Let d be the current store and θ be the substitution $[\overline{t}/\overline{x}]$ for some list of terms $\overline{t}$. If d entails $d' \otimes c\theta$ for some d' (i.e., $d \vdash d' \otimes c\theta$), the ask agent $\forall \overline{x}(c \to P)$ *consumes* $c\theta$ and executes $P\theta$ under the new store d'. If c (the *guard*) cannot be deduced from d, the ask agent blocks until more information is added to the store. If the sequence of variables $\overline{x}$ is empty then $\forall \overline{x}(c \to P)$ is written as $c \to P$.

Parallel Composition. $P \parallel Q$ stands for the interleaved parallel execution of agents P and Q, possibly communicating through shared variables in the store. Given a finite set of indexes $I = \{1, 2, ..., n\}$, instead of $P_1 \parallel P_2 \parallel ... \parallel P_n$, we write $\prod_{i \in I} P_i$.

Locality. The agent $\exists x(P)$ behaves like P and binds the variable x to be local to it.

Non-deterministic choice. The process $P_1 + ... + P_n$ non-deterministically chooses one P_i for execution whenever P_i can evolve (one-step guarded choice). The chosen alternative precludes the others. We assume here that each P_i is an ask agent. Hence, the agent $\sum_{i \in I} \forall \overline{x_i}(c_i \to P_i)$ evolves into $P_j[\overline{t_j}/\overline{x_j}]$ whenever the store entails $c_j[\overline{t_j}/\overline{x_j}]$ for some $j \in I$. Otherwise, the agent blocks until more information is added to the store.

Procedure Calls. Assume a process declaration:

$$p(\overline{x}) \triangleq P$$

where all free variables of P are in the set of pairwise distinct variables $\overline{x}$. The agent $p(\overline{y})$ evolves into $P[\overline{y}/\overline{x}]$.

We assume that "⊗" has a higher precedence than "→", hence $c_1 \otimes c_2 \to c_1' \otimes c_2'$ should be read as $(c_1 \otimes c_2) \to (c_1' \otimes c_2')$. Furthermore, "!" has a tighter binding than ⊗ so we understand $!\, c_1 \otimes c_2$ as $(!\, c_1) \otimes c_2$. For the rest of the operators we shall explicitly use parenthesis to avoid confusions.

In the following example we show how lcc agents evolve. We shall use $\langle P, c \rangle \to \langle P', c' \rangle$ to denote that the agent P under store c evolves into the agent P' producing the store c'. The reader may refer to [7] for a complete account of the lcc operational semantics.

Example 2.1 (Consuming Permissions)**.** *Let's assume that we have a constraint system with a ternary predicate* $\texttt{ref}(\cdot)$*, constant symbols* unq *and* shr *and equipped with the axiom:* $\Delta = \texttt{ref}(x, o, \texttt{unq}) \vdash \texttt{ref}(x, o, \texttt{shr})$*. Let's assume also a process* $R = P \parallel Q$ *such that*

$$
\begin{array}{rcl}
P & = & \texttt{ref}(x, o, \texttt{unq}) \\
Q & = & \forall y(\texttt{ref}(x, y, \texttt{shr}) \to Q')
\end{array}
$$

From the initial store $\top$ *(true),* Q *cannot deduce its guard and it remains blocked. Hence,* P *evolves by executing the tell agents* $\texttt{ref}(x, o, \texttt{unq})$*:*

$$\langle R, \top \rangle \to \langle Q, \top \otimes \texttt{ref}(x, o, \texttt{unq}) \rangle$$

Afterwards, the store $\top \otimes \texttt{ref}(x, o, \texttt{unq})$ *is strong enough to entail the guard of* Q *by using the axiom* Δ*. We thus observe the following transition:*

$$\langle Q, \top \otimes \texttt{ref}(x, o, \texttt{unq}) \rangle \to \langle Q'[o/y], \top \rangle$$

Roughly speaking, P *adds to the store the information required to state that* x *points to* o *and has a unique permission to* o *(i.e.,* $\texttt{ref}(x, o, \texttt{unq})$*). By using* Δ*, from* $\texttt{ref}(x, o, \texttt{unq})$ *we can deduce* $\texttt{ref}(x, o, \texttt{shr})$*, i.e., the unique permission of* x *can be downgraded to a share permission on* o*. Thereafter,* Q consumes *this information, leading to the store* $\top$ *where the agent* $Q'[o/y]$ *is executed.* □

We finish this section by introducing the derived operator $P; Q$ that delays the execution of Q until the "end" of the execution of P. This will be useful for the model we present in the forthcoming sections. Let z, w, w' be variables that do not occur either in P or in Q and $\texttt{sync}(\cdot)$ be an uninterpreted predicate symbol. The pro-

cess $P;Q$ can be defined as $\exists z(\mathcal{C}[\![P]\!]_z \parallel \texttt{sync}(z) \rightarrow Q)$ where

$$\begin{array}{lcl}
\mathcal{C}[\![c]\!]_z & = & c \otimes \texttt{sync}(z) \\
\mathcal{C}[\![\forall y(c \rightarrow P)]\!]_z & = & \forall y(c \rightarrow \mathcal{C}[\![P]\!]_z) \\
\mathcal{C}[\![P \parallel R]\!]_z & = & \exists w, w'(\mathcal{C}[\![P]\!]_w \parallel \mathcal{C}[\![R]\!]_{w'} \parallel \\
 & & \quad \texttt{sync}(w) \otimes \texttt{sync}(w') \rightarrow \texttt{sync}(z)) \\
\mathcal{C}[\![P + R]\!]_z & = & \mathcal{C}[\![P]\!]_z + \mathcal{C}[\![R]\!]_z \\
\mathcal{C}[\![\exists y(P)]\!]_z & = & \exists y(\mathcal{C}[\![P]\!]_z) \\
\mathcal{C}[\![p(\overline{x})]\!]_z & = & p(\overline{x}, z)
\end{array}$$

Intuitively $\mathcal{C}[\![P]\!]_z$ adds the constraint $\texttt{sync}(z)$ when it terminates. Then, the ask agent $\texttt{sync}(z) \rightarrow Q$ reduces to Q. Notice for example that in a parallel composition $P \parallel R$, we wait for both P and R to finish and then, the constraint $\texttt{sync}(z)$ is emitted. As we shall see, we only use calls and process definitions when modeling aliasing of variables and Æminium constructs and methods declarations. Hence, in Section 3.2 we shall rewrite the signature of a process definition $p(\overline{y})$ as $p(\overline{y}, w) \triangleq P$. Then, the call $p(\overline{x}, z)$ evolves into $P[\overline{x}/\overline{y}, z/w]$ that later adds the constraint $\texttt{sync}(z)$ when needed to synchronize with the rest of the processes.

3. A LCC Interpretation of AP

Our `lcc` interpretation of access permissions in Æminium programs assumes a constraint system with the following axioms, predicate and constant symbols:

Permissions: We assume the set of constant symbols $\texttt{PER} = \{\texttt{unq}, \texttt{shr}, \texttt{imm}, \texttt{none}\}$ in order to represent the permissions introduced in Section 2.1.

References and Fields: We use the predicate symbol $\texttt{ref}(x, o, p)$ (x points to object o with permission $p \in \texttt{PER}$), $\texttt{field}(x, o, field)$ (x points to $o.field$), $\texttt{sync}(z)$ (synchronizing on variable z) and $\texttt{ct}(o, n)$ (there are n references pointing to o). For the last constraint, we also assume the constant $\mathbf{0}$ (zero) and the successor function $s(\cdot)$. Furthermore, we assume the constants nil (null reference) and $cname\_fname$ for each field "$fname$" of class "$cname$".

Non-logical axioms: We assume the following axioms:

$$\begin{array}{lcl}
\texttt{downgrade}_1 & : & \texttt{ref}(x, o, \texttt{unq}) \vdash \texttt{ref}(x, o, \texttt{shr}) \\
\texttt{downgrade}_2 & : & \texttt{ref}(x, o, \texttt{unq}) \vdash \texttt{ref}(x, o, \texttt{imm}) \\
\texttt{upgrade}_1 & : & \texttt{ref}(x, o, \texttt{shr}) \otimes \texttt{ct}(o, s(\mathbf{0})) \\
 & & \vdash \texttt{ref}(x, o, \texttt{unq}) \otimes \texttt{ct}(o, s(\mathbf{0})) \\
\texttt{upgrade}_2 & : & \texttt{ref}(x, o, \texttt{imm}) \otimes \texttt{ct}(o, s(\mathbf{0})) \\
 & & \vdash \texttt{ref}(x, o, \texttt{unq}) \otimes \texttt{ct}(o, s(\mathbf{0}))
\end{array}$$

The axiom $\texttt{downgrade}_1$ (resp. $\texttt{downgrade}_2$) transforms a unique permission into a share (resp. immutable) permission. The axiom $\texttt{upgrade}_1$ (resp. $\texttt{upgrade}_2$) builds a unique permission from a share (resp. immutable) permission. Hence, to be able to upgrade a permission to unique, the reference x needs to be the unique reference with share or immutable permission to the pointed object o. Conversions from share permissions into immutable and vice versa require first to upgrade the permission to unique and then, apply the appropriate downgrade axiom.

3.1 Modeling Statements.

We interpret Æminium statements through the function $\mathcal{S}[\![s]\!]_z$ that given a statement s returns an `lcc` agent that synchronizes with the rest of the program by adding the constraint $\texttt{sync}(z)$ to the store. We assume (by renaming variables if necessary) that z does not occur in s. In the following we define $\mathcal{S}[\![s]\!]_z$ for each type of statement in Figure 2.

Local variables in Æminium are defined as local agents in `lcc`. The local variable x points to nil with no permissions.

$$(\mathrm{R_{LOC}})\ \ \mathcal{S}[\![\texttt{let } x \texttt{ in } s \texttt{ end}]\!]_z = \exists x(\texttt{ref}(x, nil, \texttt{none}); \mathcal{S}[\![s]\!]_z)$$

For the **assignment** $x := y$, we define the rule:

$$(\mathrm{R_{ALIAS}})\quad \mathcal{S}[\![x := y]\!]_z \ = \texttt{assg}(x, y, z)$$

where

$$\begin{array}{l}
\texttt{assg}(x, y, z) \triangleq \ \ \texttt{drop}(x); \texttt{gain}(x, y); \texttt{sync}(z) \\
\texttt{drop}(x) \ \stackrel{\texttt{def}}{=}\ \ \forall o, n((\texttt{ref}(x, nil, \texttt{none}) \rightarrow \top) + \\
\qquad \sum_{p \in \texttt{PER} \setminus \{\texttt{none}\}} \texttt{ref}(x, o, p) \otimes \texttt{ct}(o, s(n)) \rightarrow \texttt{ct}(o, n)) \\
\texttt{gain}(x, y) \ \stackrel{\texttt{def}}{=} \\
\quad \texttt{ref}(y, nil, \texttt{none}) \rightarrow \texttt{ref}(x, nil, \texttt{none}) \otimes \texttt{ref}(y, nil, \texttt{none}) \\
+\ \forall o, n((\texttt{ref}(y, o, \texttt{unq}) \otimes \texttt{ct}(o, s(\mathbf{0})) \rightarrow \\
\qquad \texttt{ref}(y, o, \texttt{shr}) \otimes \texttt{ref}(x, o, \texttt{shr}) \otimes \texttt{ct}(o, s(s(\mathbf{0})))) \\
\qquad + (\texttt{ref}(y, o, \texttt{shr}) \otimes \texttt{ct}(o, n) \rightarrow \\
\qquad \texttt{ref}(y, o, \texttt{shr}) \otimes \texttt{ref}(x, o, \texttt{shr}) \otimes \texttt{ct}(o, s(n))) \\
\qquad + (\texttt{ref}(y, o, \texttt{imm}) \otimes \texttt{ct}(o, n) \rightarrow \\
\qquad \texttt{ref}(y, o, \texttt{imm}) \otimes \texttt{ref}(x, o, \texttt{imm}) \otimes \texttt{ct}(o, s(n)))
\end{array}$$

Here, the variable x loses its permission to the pointed object o, and the object o has one less reference pointing to it (Definition `drop`[1]). Thereafter, x and y point to the same object and the permission of y is split between x and y (Definition `gain`) as follows: if y has a unique permission to o, this permission is split into two share permissions, one for x and one for y. If y has a share (resp. immutable) permission to o, then both x and y will have a share (resp. immutable) permission to o after the assignment. Recall that ask agents consume their guard when evolving. Therefore, we add back the permission for y in the right-hand side of the rule. Finally, once the permission to y is split, the constraint $\texttt{sync}(z)$ is added to the store to synchronize with the rest of the program.

If the variable x points to the object o of class $cname$, then the field $fname$ of o can be accessed by the variable u whenever $\texttt{field}(u, o, cname\_fname)$ holds. Intuitively, u points to $o.fname$ and then a constraint $\texttt{ref}(u, o', p)$ enforces $o.fname$ to point to o' with permission p. As we shall show later, the model of constructors adds the constraint $!\,\texttt{field}(u, o, cname\_fname)$ to establish the connection between objects and their fields. The model of the assignment $\mathcal{S}[\![x.fname := y]\!]$ is thus obtained from that of $\mathcal{S}[\![u := y]\!]$:

$$\begin{array}{l}
(\mathrm{R_{ALIAS_F}}) \\
\mathcal{S}[\![x.f := y]\!]_z = \forall u, o, p(\ \texttt{ref}(x, o, p) \otimes \texttt{field}(u, o, cname_f) \\
\qquad\qquad \rightarrow (\texttt{ref}(x, o, p); \mathcal{S}[\![u := y]\!]_z))
\end{array}$$

The models for the statements $\mathcal{S}[\![x.fname := y.fname]\!]_z$ and $\mathcal{S}[\![x := y.fname]\!]_z$ are similar and thus omitted.

For the **composition of statements** $\{s_1\ s_2\ ...\ s_n\}$, the agent modeling s_i runs in parallel with the other agents once the agent modeling the statement s_{i-1} adds the constraint $\texttt{sync}(z_{i-1})$. After the execution of the statement s_n, the constraint $\texttt{sync}(z)$ is added to the store to synchronize with the rest of the program.

$$\begin{array}{l}
(\mathrm{R_{COMP}}) \\
\quad \mathcal{S}[\![\{s_1 \ldots s_i \ldots s_n\}]\!]_z \ = \ \exists z_1, \ldots z_n(\mathcal{S}[\![s_1]\!]_{z_1} \parallel \\
\qquad\qquad \texttt{sync}(z_1) \rightarrow \mathcal{S}[\![s_2]\!]_{z_2} \parallel \ldots \parallel \\
\qquad\qquad \texttt{sync}(z_{n-1}) \rightarrow \mathcal{S}[\![s_n]\!]_{z_n} \parallel \\
\qquad\qquad \texttt{sync}(z_n) \rightarrow \texttt{sync}(z))
\end{array}$$

Method calls and **Object instantiation**. For the sake of simplicity, we write methods and constructors using functional nota-

[1] Definitions ($\stackrel{\texttt{def}}{=}$) must be understood as shorthands.

tion rather than object-oriented notation. For instance, $x.meth(\overline{y})$ is written as $cname\_meth(x, \overline{y})$ when x is an object of type $cname$. Similarly, the expression $x := \texttt{new}\ cname(\overline{y})$ is written as $cname\_cname(x, \overline{y})$. As we shall see, for each method of the form $meth(x, \overline{y})$ in class $cname$, we shall generate a process definition $cname\_meth(x, \overline{y}, z) \triangleq P$. The Æminium statement $cname\_meth(x, \overline{y})$ is then modeled as the `lcc` call $cname\_meth(x, \overline{y}, z)$. This thus triggers the execution of the body of the method. Notice that we add the variable z as last parameter to be able to synchronize with the rest of the program.

$$(\mathrm{R_{CALL}}) \quad \mathcal{S}[\![x.meth(\overline{y})]\!]_z \quad = \quad cname\_meth(x, y_1, .., y_n, z) \text{ if } x \text{ is of type } cname$$

The model of an object initialization is defined similarly:

$$(\mathrm{R_{NEW}})\ \mathcal{S}[\![x := \texttt{new}\ cname(\overline{y})]\!]_z = cname\_cname(x, \overline{y}, z)$$

3.2 Modeling Class Definitions.

The model of method declarations and constructors is given by the function $\mathcal{D}[\![\cdot]\!]$. Hence, a **method definition** M_D of the class $cname$ of the form

$$meth(cname\ x, \overline{class_y\ y})\ \ p(x), \overline{p(y)} \Rightarrow p'(x), \overline{p'(y)}\ \{s\}$$

is modeled as a process definition:

$$(\mathrm{R_{MDEF}})\ \ \mathcal{D}[\![M_D]\!] = cname\_meth(x, \overline{y}, z) \triangleq P_M$$

Recall that the first parameter x of the method represents the object caller `this` and the last parameter z is used for synchronization. The body of the definition P_M models the behavior of the method as follows:

$$P_M \stackrel{\text{def}}{=} \forall \overline{o}, o_t, \overline{n}, n_t(\texttt{consume}; \exists \overline{y'}, x'(\texttt{params}\ ; \texttt{sync}(z); P_B))$$

where $m = |\overline{y}|$ is the number of parameters of the method and $|\overline{o}| = |\overline{n}| = m$. The process P_M first consumes the required permissions from the parameters $\overline{y}$ and from the caller x. If the required permission is share or immutable, those permissions are restored to allow concurrent executions in the environment that called the method. Unique and none permissions are consumed to later be *transferred* to the body of the method:

$$\begin{array}{lll}
\texttt{consume} & \stackrel{\text{def}}{=} & \prod_{i \in 1..m} \texttt{consume}_{y_i}; \texttt{consume}_x \\
\texttt{consume}_{y_i} & \stackrel{\text{def}}{=} & \texttt{ref}(y_i, o_i, p_i) \otimes \texttt{ct}(o_i, n_i) \rightarrow \\
 & & \texttt{ref}(y_i, o_i, p_i) \otimes \texttt{ct}(o_i, s(n_i)) \text{ if } p_i \in \{\texttt{shr}, \texttt{imm}\} \\
\texttt{consume}_{y_i} & \stackrel{\text{def}}{=} & \texttt{ref}(y_i, o_i, p_i) \rightarrow \top \text{ if } p_i \in \{\texttt{unq}, \texttt{none}\}
\end{array}$$

Definition $\texttt{consume}_x$ is similar to that of $\texttt{consume}_{y_i}$ but it considers the variable x, the object o_t and the permission p.

Once the permissions are consumed according to the signature of the method, the agent P_M creates local variables $\overline{y'}$ and x' to replace the formal parameters ($\overline{y}$) and the caller (x) by the actual parameters:

$$\texttt{params} \stackrel{\text{def}}{=} \texttt{ref}(x', o_t, p) \otimes \bigotimes_{i \in 1..m} \texttt{ref}(y'_i, o_i, p_i)$$

At this point, P_M adds $\texttt{sync}(z)$ to release the program control. Thereafter the body of the method can be executed. This is done by modeling the statement s as the agent P_B where $\widehat{s}$ denotes s after replacing y_i by y'_i and x by x':

$$P_B \stackrel{\text{def}}{=} \exists z'(\mathcal{S}[\![\widehat{s}]\!]_{z'}; \texttt{sync}(z') \rightarrow (\texttt{r\_env}(x : p, x' : p') \parallel \prod_{i \in 1..m} \texttt{r\_env}(y_i : p_i, y'_i : p'_i)))$$

Once the execution of s releases the control (i.e., it adds $\texttt{sync}(z')$ to the store), the references and permissions of the local variables created to handle the parameters are consumed and restored to the environment according to:

$$\begin{array}{lll}
\texttt{r_env}(x : p, x' : p') & \stackrel{\text{def}}{=} & \forall o', n(\texttt{ref}(x', o', p') \otimes \texttt{ct}(o', s(n)) \rightarrow \\
 & & \quad \texttt{ct}(o', n)) \quad \text{if } p, p' \in \{\texttt{imm}, \texttt{shr}\} \\
\texttt{r_env}(x : p, x' : p') & \stackrel{\text{def}}{=} & \forall o'(\texttt{ref}(x', o', p') \rightarrow \\
 & & \quad \texttt{ref}(x, o', p')) \quad \text{if } p \in \{\texttt{unq}, \texttt{none}\} \\
\texttt{r_env}(x : p, x' : p') & \stackrel{\text{def}}{=} & \forall o, n, o'((\texttt{ref}(x, o, p) \otimes \texttt{ct}(o, s(n)) \rightarrow \\
 & & \quad \texttt{ct}(o, n)); \\
 & & \quad \texttt{ref}(x', o', p') \rightarrow \texttt{ref}(x, o', p')) \\
 & & \quad \text{if } p \in \{\texttt{shr}, \texttt{imm}\}, p' \in \{\texttt{unq}, \texttt{none}\}
\end{array}$$

Let us give some intuition about the cases considered in the definitions above. Recall that `consume` *duplicates* the `shr` and `imm` permissions for the variables internal to the method. Then, we only need to consume such permissions and decrease the number of references pointing to object o'. As for `unq` and `none` as input permissions, `consume` *transfers* such permissions to the local variables and *consumes* the external references. Then, `r_env` needs to restore the external reference and consume the local one (the number of references pointing to o' remains the same). When the method changes the input permission from share or immutable into a unique or none, we need to *consume* first the external reference. Then, we *transfer* the internal permission and reference to the external variable.

A **constructor** C_D of the form

$$cname(cname\ x, \overline{class_y\ y})\ \ \texttt{none}(x), \overline{p(y)} \Rightarrow p'(x), \overline{p'(y)}\ \{s\}$$

is modeled similarly as a method definition:

$$(\mathrm{R_{CDEF}})\mathcal{D}[\![\mathrm{C_D}]\!] = cname(x, \overline{y}, z) \triangleq P_C$$

$$\begin{array}{lll}
P_C & \stackrel{\text{def}}{=} & \forall \overline{o}(\texttt{consume}'; \\
 & & \quad \exists \overline{y'}, x', o_{new}(\texttt{params}'; \\
 & & \quad\quad \exists \overline{u}(\texttt{fields-init}\ ; \\
 & & \quad\quad\quad \exists z'(\mathcal{S}[\![\widehat{s}]\!]_{z'}; \texttt{sync}(z') \rightarrow (\texttt{r_env}(x : p, x' : p') \parallel \\
 & & \quad\quad\quad\quad \prod_{i \in 1..m} \texttt{r_env}(y_i : p_i, y'_i : p'_i))))) \\
 & & \quad ; \texttt{sync}(z))
\end{array}$$

Here $\texttt{consume}'$ is similar to `consume` but with $o_t = nil$, i.e., x in $x := \texttt{new}\ cname(\overline{y})$ is restricted to be a null reference. Definition $\texttt{params}'$ is similar to `params` except that it considers $p = \texttt{unq}$, i.e. x' has a unique permission to o_{new}. Furthermore, $\texttt{params}'$ adds $\texttt{ct}(o_{new}, s(\mathbf{0}))$ to the store. Class fields are initialized to nil and the link between the variable u_i and the field $o_{new}.f_i$ is established:

$$\begin{array}{l}
\texttt{fields-init} \stackrel{\text{def}}{=} \\
\quad !\texttt{field}(u_1, o_{new}, cname_f_1) \otimes \texttt{ref}(u1, nil, \texttt{none}) \otimes ... \otimes \\
\quad !\texttt{field}(u_k, o_{new}, cname_f_k) \otimes \texttt{ref}(u_k, nil, \texttt{none})
\end{array}$$

Finally, notice that the synchronization constraint $\texttt{sync}(z)$ is added only in the end of the rule since the constructor needs to be fully executed before returning the new reference.

The following example shows how the proposed model works.

Example 3.1 (Access Permission Flow)**.** *Assume the class definitions stats and collection in Figure 1 and the following main body written in functional notation.*

```
1  let collection c, stats s in
2    collection_collection(c);  //c := new collection()
3    stats_stats(s);  //s := new stats()
4    collection_compStats(c, s);  //c.compStats(s)
5    collection_removeDuplicates(c); //c.rDup() end
```

The `lcc` *agent modeling the statement in line* 2 *performs the call* $collection\_collection(c, z_1)$*, which triggers the execution of the*

body of the constructor collection (see Rules $\mathrm{R}_{\mathrm{CDEF}}$ *and* $\mathrm{R}_{\mathrm{CALL}}$ *). Variable* z_1 *is the local variable used to synchronize with the rest of the program (see Rule* $\mathrm{R}_{\mathrm{COMP}}$*). Once the agents modeling the statements in lines* 2 *and* 3 *are executed, the store below is observed.*

$$\exists c, s, o_c, o_s(\mathtt{ref}(c, o_c, \mathtt{unq}) \otimes \mathtt{ref}(s, o_s, \mathtt{unq}) \otimes \mathtt{ct}(o_c, s(\mathbf{0})) \otimes \mathtt{ct}(o_s, s(\mathbf{0})))$$

Hence, c *(resp.* s*) points to* o_c *(resp.* o_s*) with a unique permission. Since* $collection\_compStats(c, s)$ *requires* c *to have an immutable permission to* o_c*, the axiom* $downgrade_1$ *is used to entail the guard of* `consume` *in the definition of the method (see Rule* $\mathrm{R}_{\mathrm{MDEF}}$*). Let* c' *be the representation of* c *inside the method (see* `params` *in Rule* $\mathrm{R}_{\mathrm{MDEF}}$*). We notice that when the body of the method is being executed, both* c *and* c' *have an immutable permission to* o_c*. Before executing the body of method* $compStats$*, the constraint* $\mathtt{sync}(z_1)$ *is added so as to allow possibly concurrent executions in the main body. The agent modeling the statement in line* 5 *can be then executed. However, this call requires* c *to have a unique permission to* o_c *which is not possible since the axiom* $upgrade_1$ *requires that* c *is the sole reference to* o_c*. Hence, the guard* `consume` *for this call is delayed (synchronized) until the permission on* c' *is consumed and restored to the environment (see definition* `r_env`*). We then observe that statements in lines* 4 *and* 5 *are executed sequentially due to the way permissions evolve.* □

3.3 The Model as a Runnable Specification

`ccp`-based models can be regarded as runnable specifications, and so we can observe how permissions evolve during program execution by running the underlying `lcc` model. We implemented an interpreter of `lcc` on top of the Mozart system (`http://www.mozart-oz.org/`). This interpreter uses records (Mozart data structures) to represent `lcc` linear constraints. The store was modeled as a multiset of records, and the entailment of constraints for universally quantified asks was implemented via record unification. On top of this interpreter, we implemented a parser that takes an Æminium program and generates the corresponding `lcc` agents. The `lcc` agent is then executed and a program trace is generated. The interpreter and the parser have been integrated into Alcove (Æminium Linear COnstraints VErifier) LCC Animator, a PHP application freely available at `http://escher.puj.edu.co/~caolarte/alcove/`. The URL further includes the examples presented in this section.

Example 3.2 (Trace of Access Permissions)**.** *The program in Example 3.1 generates the following trace:*

```
[init(collection_collection [c1 z9])]
[running(collection_collection [c1 z9])]
[init(stats_stats [s2 z10])]
[running(stats_stats [s2 z10])]
[init(collection_compstats [c1 s2 z11])]
[running(collection_compstats [c1 s2 z11])]
[init(collection_removeduplicates [c1 z13])
 running(collection_compstats [c1 s2 z11])]
[running(collection_removeduplicates [c1 z13])]

c1(obj:ot16 objfields:none per:unq)
s2(obj:ot27 objfields:none per:unq)
```

Output `init(collection_collection [c1 z9])` *represents the call to a method (recall that parameter* z *is used for synchronization purposes). If a method is currently being executed, the constraint* `running(collection_collection [c1 z9])` *is present in the store. Notice that the execution of the method* $collection\_removeduplicates$ *is delayed until the end of the execution of* $collection\_compstats$ *(i.e., the store does not contain simultaneously both* `running(collection_compstats)` *and* `running(collection_removeduplicates))` *as explained in Example 3.1. The last two lines of the trace show that both* c *and* s *ends with a unique permission to objects* `ot16` *and* `ot27` *respectively* [2] □

Example 3.3 (Deadlock Detection)**.** *Let us assume now the class definitions in Figure 1 and the following* `main`*:*

```
1  let collection c, stats s, stats svar in
2    collection_collection(c);  //c := new collection()
3    stats_stats(s);  //s := new stats()
4    svar := s;
5    collection_compStats(c, s);  //c.compStats(s)
6  end
```

This code aliases $svar$ *and* s *after the assignment* $svar := s$*, so that they share the same permission afterwards. Therefore,* s *cannot recover the unique permission to execute the statement* $collection\_compStats(c, s)$*, thus leading to a permission deadlock. This bug is detected by Alcove:*

```
[init(collection_collection [c1 z10])]
[running(collection_collection [c1 z10])]
[init(stats_stats [s2 z11])]
[running(stats_stats [s2 z11])]
[init(collection_compstats [c1 s2 z14])]
c1(obj:ot17 objfields:none per:imm)
svar9(obj:ot28 objfields:none per:shr)
s2(obj:ot28 objfields:none per:shr)
Error: Permissions for collection_compstats(c1 s2 z14)
       could not be obtained.
```

We notice that in the trace above, the call to the method $compstats$ *is invoked (*`init`*) but the method was not executed (*`running`*). Furthermore, both* s *and* $svar$ *have a share permission on the pointed object.* □

4. Verification Techniques

Besides playing the role of executable specifications, `ccp`-based models can be declaratively interpreted as formulas in logic (see e.g., [17]). This section provides additional mechanisms and tools for verifying properties of access-permission based programs. More concretely, we take the `lcc` agents generated by the Alcove LCC Animator and translate them into a linear logic (LL) formula. Then, a property specified in LL is verified with the Alcove LL Prover, a bespoken theorem prover implemented on top of λ-Prolog [14] based on the prover described in [9].

4.1 Agents as Formulas

In `lcc`, processes are not only agents that evolve according to the rules of the underlying operational semantics, but also are formulas in linear logic [8]. The logical interpretation of `lcc` is defined with the aid of a function $\mathcal{L}[\![\cdot]\!]$ defined as [7]:

$$\begin{array}{lcl} \mathcal{L}[\![c]\!] & = & c \\ \mathcal{L}[\![p(\overline{x})]\!] & = & p(\overline{x}) \\ \mathcal{L}[\![P \parallel Q]\!] & = & \mathcal{L}[\![Q]\!] \otimes \mathcal{L}[\![P]\!] \\ \mathcal{L}[\![P + Q]\!] & = & \mathcal{L}[\![Q]\!] \& \mathcal{L}[\![P]\!] \\ \mathcal{L}[\![\forall \overline{x}(c \rightarrow P)]\!] & = & \forall \overline{x}(c \multimap \mathcal{L}[\![P]\!]) \\ \mathcal{L}[\![\exists x(P)]\!] & = & \exists x(\mathcal{L}[\![P]\!]). \end{array}$$

where $\&$ is the linear additive conjunction and $\multimap$ is the linear implication. The first step of our approach for the verification of programs consists in interpreting the `lcc` model in Section 3 as a LL formula according to function $\mathcal{L}[\![\cdot]\!]$. Furthermore, process definitions of the form $p(\overline{x}) \triangleq P$ (i.e., assignment and constructor and method definitions) are transformed into a LL clause

[2] The numbers that follow the variable names are generated each time a local variable is created to avoid clash of names.

$\forall \overline{x}.p(\overline{x}) \multimap P$. We shall call these clauses *definition clauses* and they are stored together with the axioms of the constraint system (upgrade and downgrade):

$$\begin{array}{l}
\mathtt{ref}(x,o,\mathtt{unq}) \multimap \mathtt{ref}(x,o,\mathtt{shr}).\\
\mathtt{ref}(x,o,\mathtt{unq}) \multimap \mathtt{ref}(x,o,\mathtt{imm}).\\
\mathtt{ref}(x,o,\mathtt{shr}) \otimes \mathtt{ct}(o,s(\mathbf{0})) \multimap \mathtt{ref}(x,o,\mathtt{unq}) \otimes \mathtt{ct}(o,s(\mathbf{0})).\\
\mathtt{ref}(x,o,\mathtt{imm}) \otimes \mathtt{ct}(o,s(\mathbf{0})) \multimap \mathtt{ref}(x,o,\mathtt{unq}) \otimes \mathtt{ct}(o,s(\mathbf{0})).
\end{array}$$

into a theory Δ. Example 4.1 illustrates this translation. Observe that, in what follows, we present a simplified version of the translation where the empty synchronizations were omitted.

Example 4.1 (Agents as formulas)**.** *Assume the program in Example 3.3. The predicate $collection\_collection(x,z)$ for the constructor is built from Rule* $\mathrm{R_{CDEF}}$*, giving rise to the following (universally quantified) definition clause:*

$$\begin{array}{r}
col_collection(x,z) \multimap \exists w_1(\mathtt{ref}(x,nil,\mathtt{none}) \multimap \mathtt{sync}(w_1) \otimes\\
\mathtt{sync}(w_1) \multimap \exists x', o_{new}, w_2(\mathtt{ref}(x', o_{new}, \mathtt{unq}) \otimes \mathtt{ct}(o_{new}, s(\mathbf{0})) \otimes\\
\mathtt{sync}(w_2) \quad \otimes\\
\mathtt{sync}(w_2) \multimap \exists w_3 \forall o'(\mathtt{ref}(x', o', \mathtt{unq}) \multimap \mathtt{ref}(x, o', \mathtt{unq}) \otimes\\
\mathtt{sync}(w_3) \quad \otimes\\
\mathtt{sync}(w_3) \multimap \mathtt{sync}(z))))).
\end{array}$$

The interpretation for methods is obtained similarly by following the rule $\mathrm{R_{MDEF}}$.

The assignment of variables is encoded by the predicate $assg(x,y,z)$ resulting from the translation of the Rule $\mathrm{R_{ALIAS}}$*:*

$$\begin{array}{l}
assg(x,y,z) \multimap\\
\exists z_1, z_2(\forall o, n((\mathtt{ref}(x,o,\mathtt{none}) \multimap \top \otimes \mathtt{sync}(z_1) \&\\
\qquad \mathtt{ref}(x,o,\mathtt{unq}) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1) \&\\
\qquad \mathtt{ref}(x,o,\mathtt{shr}) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1) \&\\
\qquad \mathtt{ref}(x,o,\mathtt{imm}) \otimes \mathtt{ct}(o,s(n)) \multimap \mathtt{ct}(o,n) \otimes \mathtt{sync}(z_1)) \otimes\\
\\
\mathtt{sync}(z_1) \multimap \mathtt{ref}(y,nil,\mathtt{none}) \multimap\\
\qquad \mathtt{ref}(x,nil,\mathtt{none}) \otimes \mathtt{ref}(y,nil,\mathtt{none}) \otimes \mathtt{sync}(z_2)) \&\\
\qquad (\forall o, n(\mathtt{ref}(y,o,\mathtt{unq}) \otimes \mathtt{ct}(o,s(\mathbf{0})) \multimap \mathtt{ref}(y,o,\mathtt{shr}) \otimes\\
\qquad \mathtt{ref}(x,o,\mathtt{shr}) \otimes \mathtt{ct}(o,s(s(\mathbf{0}))) \otimes \mathtt{sync}(z_2) \&\\
\qquad \mathtt{ref}(y,o,\mathtt{shr}) \otimes \mathtt{ct}(o,n) \multimap \mathtt{ref}(y,o,\mathtt{shr}) \otimes\\
\qquad \mathtt{ref}(x,o,\mathtt{shr}) \otimes \mathtt{ct}(o,s(n)) \otimes \mathtt{sync}(z_2) \&\\
\qquad \mathtt{ref}(y,o,\mathtt{imm}) \otimes \mathtt{ct}(o,n) \multimap \mathtt{ref}(y,o,\mathtt{imm}) \otimes\\
\qquad \mathtt{ref}(x,o,\mathtt{imm}) \otimes \mathtt{ct}(o,s(n)) \otimes \mathtt{sync}(z_2))) \otimes\\
\qquad \mathtt{sync}(z_2) \multimap \mathtt{sync}(z)).
\end{array}$$

Hence, for this example, the theory Δ would contain the definition clauses for $collection\_collection$, $stats\_stats$, $assig$ and $collection\_compStats$, together with axioms for upgrading and downgrading permissions.

Let A be the `lcc` *agent related to the main program. Then,*

$$\begin{array}{rl}
F = & \exists c, s, svar, z, z_1, z_2, z_3, z_4, z_5(\mathtt{ref}(c,nil,\mathtt{none}) \otimes\\
& \mathtt{ref}(s,nil,\mathtt{none}) \otimes \mathtt{ref}(svar,nil,\mathtt{none}) \otimes \mathtt{sync}(z_1) \otimes\\
& \quad \mathtt{sync}(z_1) \multimap collection_collection(c,z_2) \otimes\\
& \quad \mathtt{sync}(z_2) \multimap stats_stats(s,z_3) \otimes\\
& \quad \mathtt{sync}(z_3) \multimap assig(svar,s,z_4) \otimes\\
& \quad \mathtt{sync}(z_4) \multimap collection_compStats(c,s,z_5) \otimes\\
& \mathtt{sync}(z_5) \multimap \mathtt{sync}(z)).
\end{array}$$

corresponds to $\mathcal{L}[\![A]\!]$ (see Rules $\mathrm{R_{COMP}}$*,* $\mathrm{R_{NEW}}$*,* $\mathrm{R_{CALL}}$ *and* $\mathrm{R_{ALIAS}}$ *in Section 3).* □

4.2 Linear Logic as a Framework for Verifying Access Permission Properties

Assume the translation $\mathcal{L}[\![A]\!]$ as described in Example 4.1, producing a theory Δ and a formula F. In order to verify a certain property $\mathcal{T}$, specified by a LL formula T, we test if the sequent $!\,\Delta, F \vdash T$ is provable. In this section, we will give an estimate of the complexity of finding such a proof.

First of all, observe that the fragment of ILL needed for encoding access permissions is given by the following grammar for guards G, processes P and properties T:

$$\begin{array}{rcl}
G & := & A \mid G \otimes G\\
P & := & \forall x.G \multimap P \mid P \otimes P \mid \exists x.P \mid P \& P \mid\\
 & & !(\forall \overline{x}.p(\overline{x}) \multimap P) \mid 1 \mid \top \mid !A \mid A\\
T & := & \exists x.T \mid G.
\end{array}$$

where A is an atomic formula. Notice that this grammar is well defined, since the left context in the sequent $!\,\Delta, F \vdash T$ will be formed by P formulas, the right context will have only T formulas. Besides, implications on the left can only introduce guards on the right side of a sequent and $G \subset T$[3].

We note that classical and intuitionistic provability coincide for this fragment, since the right side of sequents are composed by existentially quantified Horn clauses.

The fragment described above is undecidable in general, due to the presence of processes declarations [13]. It turns out that Æminium applications dealt in the present paper are such that process declarations $p(\overline{x})$ do not have circular recursive calls. More precisely, in an Æminium program, there is no a sequence of methods or constructs of the form $m_1, m_2, \cdots m_n$ such that m_i calls m_{i+1} and m_n calls m_1. Hence, if a method m_1 calls m_2, we can syntactically expand the body of m_2 into the body of m_1. Therefore, it is straightforward to see that provability in the resulting LL translation is decidable (see Theorem 4.1). It is worth mentioning that the analyses presented here could be enhanced in order to deal with mutual recursive calls and some types of controlled recursion, as in [16] (see more in Section 5).

Complexity Analysis. We will show now how to measure the complexity of proofs in our system. It is worth noticing that Alcove LL Prover actually uses the proposed measure as a limit on the proof search.

For reasoning about complexity of proofs in LL we need to use a proof system for it where proof search can be controlled and measured. We thus move from ILL to the *focused classical* linear logic system in *one sided* sequent style (LLF) [12][4]. In a nutshell, moving into the *classical* setting means adding the connectives ? (exponential dual to !) and $\oplus, \parr$ (additive and multiplicative versions of the disjunction) together with their neutral elements, 0 and $\bot$ respectively. *One sided* means moving from sequents of the shape $!\,\Delta, F \vdash T$ into sequents of the form $\vdash ?\Delta^{\bot}, P^{\bot}, T$, where negation is a logical connective that has only atomic scope: if B is a general formula then $B^{\bot}$ denotes the result of moving negations inward until it has only atomic scope. We shall call *literal* an atomic formula or its negation. For convenience, the clause $B \multimap C$ will be represented by the formula $B^{\bot} \parr C$.

Intuitively, the *focusing* discipline organizes proofs into two alternating phases, called *negative* and *positive* phases. In the negative phase, all (invertible) rules over the connectives of *negative polarity* ($\forall, \parr, \&, ?, \bot, \top$) are applied eagerly, while in the positive phase a formula of *positive* polarity ($\exists, \otimes, \oplus, !, 0, 1$) is focused on and its positive subformulas are eagerly introduced.

Thus, on searching for proofs in focused systems, the only nondeterministic step is the one choosing the *positive formula* to focus on from the context. This determines completely the complexity of a proof in LLF and justifies the next definitions.

[3] On examining a proof bottom-up, decomposing the implication on the sequent $\Gamma_1, \Gamma_2, B \multimap C \vdash D$ will produce the premises $\Gamma_1, C \vdash D$ and $\Gamma_2 \vdash B$. Hence it is important to guarantee that B is a T formula.

[4] As already noted, provability in the fragment used here is the same in intuitionistic and classical settings.

Definition 3 (Proof Depth)**.** *Let* Π *be a proof in LLF. The* depth *of* Π *is the maximum number of decisions over focused formulas along any path in* Π *from the root.*

Definition 4 (Degree of a positive formula)**.** *The* degree *of a positive formula is the maximum number of nested alternating polarities in it.*

The next lemma shows the relation between depth of derivations in LLF and degree of a formula. The proof is discharged by structural induction.

Lemma 4.1. *Decomposing a focused positive formula* F *of degree* n *into its literal or purely positive subformulas gives rise to a derivation of depth* $\lceil \frac{n}{2} \rceil$.

Example 4.2 (Degree of a formula)**.** *Consider the negation of the definition clause for* $collection\_collection(x, z)$ *in Example 4.1:*

$$\begin{array}{l} col\_collection(x,z) \otimes \forall w_1(\mathtt{ref}(x, nil, \mathtt{none}) \otimes \mathtt{sync}(w_1)^\perp \parr \\ \mathtt{sync}(w_1) \otimes \forall x', o_{new}, w_2(\mathtt{ref}(x', o_{new}, \mathtt{unq})^\perp \parr \mathtt{ct}(o_{new}, s(\mathbf{0}))^\perp \\ \quad \parr \mathtt{sync}(w_2)^\perp \parr \\ \quad \mathtt{sync}(w_2) \otimes \forall w_3 \exists o'(\mathtt{ref}(x', o', \mathtt{unq}) \otimes \mathtt{ref}(x, o', \mathtt{unq})^\perp \parr \\ \quad\quad \mathtt{sync}(w_3)^\perp \parr \\ \quad \mathtt{sync}(w_3) \otimes \mathtt{sync}(z)^\perp)))). \end{array}$$

The degree of such a formula is 10*. Hence the depth of decomposing the formula above into its literal or purely positive subformulas is* $10/2 = 5$. □

We will now proceed with a careful complexity analysis of all the formulas produced by the specification of Æminium programs. The calculation of the complexity is done by counting the changes of nested polarities, which are produced mostly by synchronizations.

- If $cname\_method(x, \overline{y}, z) \multimap P$ is a definition clause (DC) in Δ, its negation $cname\_method(x, \overline{y}, z) \otimes P^\perp$ is a positive formula of degree at most $11 + n + m$ where m is the length of $\overline{y}$ and n is the degree of the formula encoding the body of the constructor, i.e., $\mathcal{S}[\![\widehat{s}]\!]^\perp_{z'}$.
- If $cname(x, \overline{y}, z) \multimap P$ is a DC in Δ, its negation is a positive formula of degree at most $12 + n + m$ where m is the length of $\overline{y}$ and n is the degree of $\mathcal{S}[\![\widehat{s}]\!]^\perp_{z'}$.
- If $\mathtt{assig}(y, x, z) \multimap P$ is a DC in Δ, its negation is a positive formula of degree at most 5.
- For any formula F interpreting an Æminium main program with n statement calls, $F^\perp$ is a negative formula whose biggest positive subformula has degree at most $(2n + 1) + m$ where m is the sum of the degrees of all negated definition clauses corresponding to the statement calls in F.
- The negated upgrade (resp. downgrade) axiom is a positive formula of degree 1 (resp. degree 0).

The next theorem determines the complexity of the provability of sequents given by specification of Æminium programs.

Theorem 4.1 (Complexity)**.** *Let* Δ *be a theory containing the definition clauses for method and constructor definitions, the definition of* `assig` *and the upgrade and downgrade axioms. Let* F *be the formula interpreting the main program and* T *a formula interpreting a property to be proven. It is decidable whether or not the sequent* $\vdash ?\Delta^\perp, F^\perp, T$ *is provable. In fact, if such a sequent is provable, then its proof is bounded in LLF by the depth* $\lceil \frac{k}{2} \rceil$ *where* $k =$ *degree* $F^\perp$.

Proof. As noted before, as there are no circular recursive definitions, we may assume that the heads of definition clauses in Δ do not contain calls for other statements, i.e., the code of such calls can be directly written as part of the head. Hence, focusing over definition clauses is completely determined by the calls in $F^\perp$. Due to the synchronization procedure, proving a sequent in Æminium is equivalent to decompose its formulas completely. Therefore, the complexity of the proof of the sequent $\vdash ?\Delta^\perp, F^\perp, T$ is completely determined by the degree of $F^\perp$ since T is a purely positive formula, hence having degree 0. □

In the following, we explain our verification technique for three properties.

Deadlock Detection. Consider Example 3.3. We already showed that this code leads to a deadlock since the variable s cannot upgrade its unique permission to execute $collection\_compStats(c, s)$. We are then interested in providing a proof to the programmer showing that the code leads to a deadlock. For doing this, let Def be the definition of the method $collection\_compStats$ and the constructor $collection\_collection$, st be the statement in the `main` program and A be the `lcc` agent $A = \exists z(\mathcal{D}[\![Def]\!] \parallel (\mathcal{S}[\![st]\!]_z \parallel \mathtt{sync}(z) \rightarrow \mathtt{ok}))$. This agent adds the constraint `ok` only when the process $\mathcal{S}[\![st]\!]_z$ adds $\mathtt{sync}(z)$. According to the definition of $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{D}[\![\cdot]\!]$, this happens only when the call $collection\_compStats(c, s)$ is able to successfully consume the permissions required for the method (see Rule $\mathrm{R_{MDEF}}$). The translation $\mathcal{L}[\![A]\!]$ will give rise to the theory Δ and the formula F described in Example 4.1. Let $F' = F \otimes \mathtt{sync}(z) \multimap \mathtt{ok}$. The verification technique consists in showing that the sequent $!\Delta, F' \vdash \mathtt{ok}$ is not provable. This verification is done automatically by using Alcove-Prover, a theorem prover for LLF developed in λ-Prolog and integrated to the tool described in Section 3.3. Basically, we look for proofs with depth less or equal to 19, given by the depth of F'. If the prover fails, that means there is no proof for the sequent above.

The URL of the Alcove tool includes the output of the theorem prover and the `lcc` interpreter for this example. It is worth noticing that the `lcc` interpreter only computes a possible trace of the program while the theorem prover gives a guarantee that a certain property is verified or not by the program. The use of "animators" and provers is complementary. Existing formal models for system construction, such as the *Rodin* ([1]) tool for the event B modeling language, usually include both. The idea is that by using the animator the user gain a global understanding of the behavior of the program before attempting the proof of more precise desirable properties. This usually avoids frustrations in trying to figure out corrections of the model to discharge unproved properties.

Concurrency Analysis. Assume now the Example in Figure 1. For the method $collection\_print$ assume that we define $\mathcal{D}'[\![\cdot]\!]$ as $\mathcal{D}[\![\cdot]\!]$ but replacing $\mathtt{sync}(z)$ with "$\mathtt{sync}(z); \mathtt{begin}_{print}(z)$" and adding "$; \mathtt{begin}_{print}(z) \rightarrow \mathtt{end}_{print}(z)$" in Rule $\mathrm{R_{MDEF}}$. Similarly for method $collection\_compStats$. This will allow us to specify when a method starts its execution and when it terminates.

Let $A = \exists z(\mathcal{D}'[\![Def]\!] \parallel \mathcal{S}[\![st]\!]_z)$ where st corresponds to the main method. One can prove the linear logic sequent

$$\mathcal{L}[\![A]\!] \vdash \exists z_1, z_2(\mathtt{begin}_{print}(z_1) \otimes \mathtt{begin}_{compStats}(z_2)) \otimes \top$$

The provability of such a sequent means that the statements $collection\_print(c)$ and $collection\_compStats(c, s)$ may be executed in parallel.

Verifying a Method Specification Finally, assume that we add the field "a" in class $collection$ and the method:

$$m1()\ \ \mathtt{unq(this)} \Rightarrow \mathtt{unq(this)}\ \{\mathtt{this}.a := \mathtt{this}\}$$

Assume also that $m1$ is called in the main body. The signature of $m1$ requires that the unique permission to the caller must be restored to the environment. Nevertheless, the implementation of the method splits the unique permission into a share permissions for the field a and another for the caller (Rule $\mathrm{R}_{\mathrm{ALIAS}}$). Then, the axiom $upgrade_1$ cannot be used to recover the unique permission and the ask agent in definition `r_env` remains blocked. An analysis similar to that of deadlocks will warn the programmer about this.

5. Concluding Remarks

We presented an approach to verify programs annotated with access-permissions. We use `lcc` to verify properties related to concurrency. Hence, program statements are modeled as `lcc` agents that faithfully represent statement permissions flow. The declarative reading of `lcc` agents as formulae in Linear Logic permits the use of theorem provers to verify properties such as deadlocks, the ability to run in parallel, and whether programs are correct with respect to access permission specifications. Central to our verification approach is the synchronization mechanism based on constraints and the logical interpretation of `lcc`. Ours is certainly a novel application for `ccp` that opens a new window for the automatic verification of (object-oriented) concurrent programs.

We automated our verification approach as the Alcove tool that implements a simulator and a prover. The simulator serves to *animate* a program by observing the evolution of its permissions. The simulator issues a message if a program blocks. It is therefore a useful companion for a verifier. A good strategy for understanding the behavior of a concurrent program is to run the simulator first to observe the global program behavior and then to run the prover to verify additional properties. We used the Alcove tool to verify the examples presented in this paper and also to verify properties about ordering of method invocations for a critical zone management system (see Appendix A). The reader can find these and other examples at the Alcove tool web-site.

Related and Future work. `ccp`-based calculi have been extensively used to reason about concurrent systems. The work in [10] proposes a timed-`ccp` model for role-based access control in distributed systems. The authors combine constraint reasoning and temporal logic model-checking to reason about when a resource (e.g. a directory in a file system) can be accessed.

Languages like Æminium [18] and Plaid [19] offer a series of guarantees such as (1) absence of AP usage protocol violation at run time; (2) when a program has deterministic results and (3) whether programs are free of race conditions on the abstract state of the objects [3, 4]. Our verification technique is complementary to those works since we have shown that well-typed programs (i.e., they follow the usage protocol of AP) can lead to a blockage.

The constraint system we propose to model the downgrade and upgrade of axioms was inspired by the work of *fractional* permissions in [4] (see also [3]). *Fractional* in this setting means that an AP can be split into several more *relaxed* permissions and then joined back to form a more *restrictive* permission. For instance, a unique permission can be split into two share permissions of weight $k/2$. Therefore, to recover a unique permission, it is necessary to have two $k/2$-share permissions. The constraint system described in this paper keeps explicitly the information about the fractions by means of the predicate $\mathtt{ct}(\cdot)$.

Chalice [11] is a program verifier for OO concurrent programs that uses permissions to control memory accesses. Unlike Æminium and Plaid, concurrency in Chalice is explicitly stated by the user by means of execution threads.

AP annotations in concurrent-by-default OO languages can be enhanced with the notion of *typestates* [2, 3]. Typestates describe abstract states in the form of state-machines, thus defining a usage protocol (or *contract*) of objects. For instance, consider the class *File* with states `opened` and `closed`. The signature of the method *open* can be specified as the agent $\mathtt{unq(this)} \otimes \mathtt{closed(this)} \rightarrow \mathtt{unq(this)} \otimes \mathtt{opened(this)}$. The general idea is to verify whether a program follows correctly the usage protocol defined by the class. For example, calling the method $read$ on a `closed` file leads to an error. Typestates then impose certain order in which methods can be called. The approach our paper defines can be straightforwardly extended to deal with typestates annotations, thus widening its applicability.

The work in [18] and [15] define more specific systems and rules for access permissions to deal respectively with *group permissions* and *borrowing permissions*. A group permission represents an abstract collection of objects and allows programmers to define containers that share the same permissions to an object. The approach of borrowing permissions aims at dealing more effectively with local variable aliasing, and how permissions flow from the environment to method formal parameters. Considering these systems in Alcove amounts to refine our model of permissions in Section 3. Verification techniques should remain the same.

We intend to relax the restriction about recursion imposed on Æminium programs for obtaining a decidable analysis. More precisely, we plan to translate more involving AP based programs with controlled recursion. This can be done by finding systems with a stratified set of definition clauses, hence allowing well formed recursion in the processes declarations. This shall allow us to take into account constructors like *iterators* in the source program.

Finally, we plan to undertake a case study on the verification of a commercial multi-task threaded application that has been used for massively parallelising computational tasks [6].

Acknowledgments

This work has been partially supported by grant 1251-521-28471 from Colciencias. The work of E. Pimentel and C. Olarte has been partially carried out during their visit to the Equipe Comète, at LIX (Ecole Polytechnique). Their visits have been supported resp. by Digiteo and DGAR funds for visitors. The work of N. Cataño has been supported by the Portuguese Research Agency FCT through the CMU-Portugal program, R&D Project Aeminium, CMU-PT/SE/0038/2008.

References

[1] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.

[2] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In G. E. Harris, editor, *OOPSLA*, pages 227–244. ACM, 2008.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 301–320. ACM, 2007.

[4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2001.

[6] N. Cataño and I. Ahmed. Lightweight verification of a multi-task threaded server: A case study with the plural tool. In G. Salaün and B. Schätz, editors, *FMICS*, volume 6959 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2011.

[7] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1): 14–41, 2001.

[8] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[9] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.

[10] R. Jagadeesan, W. Marrero, C. Pitcher, and V. A. Saraswat. Timed constraint programming: a declarative approach to usage control. In P. Barahona and A. P. Felty, editors, *PPDP*, pages 164–175. ACM, 2005.

[11] K. R. M. Leino. Verifying concurrent programs with Chalice. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, page 2. Springer, 2010.

[12] C. Liang and D. Miller. A focused approach to combining logics. *Ann. Pure Appl. Logic*, 162(9):679–697, 2011.

[13] P. Lincoln, J. C. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. *Ann. Pure Appl. Logic*, 56 (1-3):239–311, 1992.

[14] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[15] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In J. Field and M. Hicks, editors, *POPL*, pages 557–570. ACM, 2012.

[16] E. Pimentel and D. Miller. On the specification of sequent systems. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2005.

[17] V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In D. S. Wise, editor, *POPL*, pages 333–352. ACM Press, 1991.

[18] S. Stork, P. Marques, and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In S. Arora and G. T. Leavens, editors, *OOPSLA Companion*, pages 933–940. ACM, 2009.

[19] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter. First-class state change in plaid. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 713–732. ACM, 2011.

A. Critical Zone Management System

Assume the class definitions for a critical zone management system in Figure 4. There are three classes, `lock`, `process` and `cs`. Each critical section has a private lock managed by an object of the class `cs`. When a process wants to enter a critical section, it tries first to invoke method `acq` of the `cs` manager. If successful, the process obtains a lock (i.e. an object of class `lock`) that it uses then to enter the critical zone. When the process wants to leave the critical zone, it invokes the method `release`. This releases ownership of the critical section lock.

Method `acq` has three parameters: `this`, the `cs` manager, `b` the process wanting to enter the critical zone and `l`, a field of `b` that will hold the lock of the `cs` supplied by the manager. Since `this` has unique permission, only one reference to the manager object can exist for `acq` to be invoked. The body of method `acq` stores the lock in `l` and a reference to the manager in field `s1` or `s2` of `b`, depending on whether the lock for `cs1` or for `cs2` is requested. Storing this reference to the manager implies that it can no longer have unique permission, so the output permission for `this` becomes shared. Moreover, `l` holds now the only reference to the private lock of the manager, so its output permission becomes unique. The effect is that field `lock1` or `lock2` of object `b` uniquely acquires the section lock. The method `enter` requires a unique permission on the lock. This ensures that only one process has a reference to the lock at any given time when entering the critical section. Method `release` restores conditions as they were before invocation to `acq`, i.e. the manager regains the unique permission and stores a unique reference to its private lock. Process fields loose the lock and the reference to the manager.

Assume now the following main code:

```
class lock {
  lock() none(this) => unq(this) {};
  enter(b) unq(this), shr(b) => unq(this), shr(b){}  }
class process{
  attrs  lock lock1,  lock lock2, cs cs1, cs cs2;
  process() none(this) => unq(this) {}  }
class cs {
  attrs  lock mylock;
  cs() none(this) => unq(this) {
     this.mylock := new lock()};
     acq1(process b, lock l) unq(this), shr(b), none(l)
                          => shr(this),shr(b),unq(l) {
        l := this.mylock;
        b.cs1 := this;
        this.mylock := null };
  acq2(process b, lock l) unq(this), shr(b), none(l)
                        => shr(this),shr(b),unq(l){
        l := this.mylock;
        b.cs2 := this;
        this.mylock := null  };
  release1(lock a, process b) shr(this),unq(a),shr(b)
                   =>unq(this),none(a),shr(b){
        this.mylock := a;
        b.cs1 := null;
        a := null };
 release2(lock a, process b) shr(this),unq(a),shr(b)
                     => unq(this),none(a),shr(b)   {
        this.mylock := a;
        b.cs2 := null;
        a := null } }
```

Figure 4. Class definitions for a critical zone management system.

```
main () {   let cs x, cs w, process y, process z in
     x:= new cs(); w := new cs();
     y := new process(); z := new process();
     x.acq1(y, y.lock1); y.lock1.enter(y);
     w.acq2(z, z.lock2); z.lock2.enter(z) ;
     x.acq1(z, z.lock1); z.lock1.enter(z);
     w.acq2(y, y.lock2); y.lock2.enter(y); }
```

where there are two section manager objects, `x` for `cs1` and `w` for `cs2`. There are also two processes, `y` and `z`. Consider the situation where `y` acquires the lock from `x` (i.e. for `cs1`) by invoking the method `acq1(x, y, y.lock1)` and enters `cs1`. Then `z` acquires the lock from `w` (i.e. for `cs2`) by invoking `acq2(w, z, z.lock2)` and enters `cs2`. Now, `z` tries to acquire the lock from `x` by invoking `acq1(x,z, z.lock1)`, but this is not possible because `x` has no longer unique permission and execution blocks. The output of Alcove for this program is:

```
Error: The end of the program could not be reached.
Error: The perm. for cs_acq1( x1 z12 __f__lock1121 z24 )
    could not be obtained.
```

Consider now the program where processes leave the critical section before attempting to acquire another lock:

```
main () {   let cs x, cs w, process y, process z in
     x:= new cs(); w := new cs();
     y := new process(); z := new process();
     x.acq1(y, y.lock1); y.lock1.enter(y);
     w.acq2(z, z.lock2); z.lock2.enter(z) ;
     x.release1(y.lock1, y);
     x.acq1(z, z.lock1); z.lock1.enter(z);
     w.release2(z.lock2, z);
     w.acq2(y, y.lock2); y.lock2.enter(y);
     x.release1(z.lock1, z);  w.release2(y.lock2, y); }
```

In this case, all invocations run without blockage and Alcove successfully finishes the analysis.